A Distributed Procedure for Computing Stochastic Expansions with `Mathematica`

Christophe Ladroue*
and
Anastasia Papavasiliou†

September 29, 2010

Abstract

The solution of a (stochastic) differential equation can be locally approximated by a (stochastic) expansion. If the vector field of the differential equation is a polynomial, the corresponding expansion is a linear combination of iterated integrals of the drivers and can be calculated using Picard Iterations. However, such expansions grow exponentially fast in their number of terms, due to their specific algebra, rendering their practical use limited.

We present a `Mathematica` procedure that addresses this issue by re-parametrising the polynomials and distributing the load in as small as possible parts that can be processed and manipulated independently, thus alleviating large memory requirements and being perfectly suited for parallelized computation. We also present an iterative implementation of the shuffle product (as opposed to a recursive one, more usually implemented) as well as a fast way for calculating the expectation of iterated Stratonovich integrals for Brownian Motion.

Key words: Rough paths, Iterated Integrals, Stochastic Expansions

AMS subject classifications: 60E20, 78M05, 60G99, 60H05

1 Motivation and mathematical background

In this section, we introduce the mathematical background and motivation for manipulating expansions and iterated integrals. The next subsection introduces the Picard procedure, a simple iterative way to derive local approximation of the solution of a differential equation. Iterated integrals are then introduced and the two are finally combined to define the expansions.

*Department of Statistics, University of Warwick, Coventry, CV4 7AL, UK.
†Department of Statistics, University of Warwick, Coventry, CV4 7AL, UK.

1.1 Motivation and notation

Consider the following differential equation:

$$dY_t = f(Y_t, \theta)dX_t \tag{1}$$

where $Y_t \in \mathbb{R}^m$, $X_t \in \mathbb{R}^n$ and $f(., \theta) \in \mathbb{R}^{m \times n}$. The parameters of the function f are collected in the variable θ. We call each $X^{(i)}$ a driver of the differential equation. We assume the functions $f_{j,i},\ i \in \{1 \dots n\},\ j \in \{1 \dots m\}$ to be polynomials. The initial value of Y_t is set to Y_0.

The objective is to derive a local approximation of the solution in terms of f, Y_0 and the iterated integrals of Y_t, *i.e.*that is integrals of the form

$$\int \dots \int_{0<u_1<\dots<u_k\leq T} dX_{u_1}^{\tau_1} \dots dX_{u_k}^{\tau_k}$$

for any word $\tau = (\tau_1, \dots, \tau_k)$ constituted of the letter $\tau_i \in \{1, \dots, n\},\ i = 1, \dots, k.$. Such expansions play an important role in the theory of rough paths, allowing one to define such a differential equation for a large class of drivers X ([3]). They have also been used recently for parameter estimation of SDE ([4]).

1.2 Picard iterations

Picard iterations provide a way for deriving local approximations of solutions of differential equation. They are defined as:

$$Y_{0,T}(0) = Y_0 - Y_0 = 0 \tag{2}$$

$$Y_{0,T}(r+1, j) = \sum_{i=1}^{n} \int f_{j,i}(Y_s(r, j))dX_s^{(i)} \tag{3}$$

$$\tag{4}$$

where $Y_{0,T}(r, j)$ is the j^{th} component of the approximation of $Y_{0,T} = \int_0^T dY_s = Y_T - Y_0$ after r iterations. Thus, the first iteration gives:

$$Y_{0,T}(1, j) = \int_0^t f_{j,1}(Y_0)dX_s^{(1)} + \dots + \int_0^t f_{j,n}(Y_0)dX_s^{(n)} \tag{5}$$

Note that if we are interested in the actual value of the solution at a time $t \in [0, T]$, its Picard approximation is given by

$$Y_t(r, j) = Y_0 + Y_{0,t}(r, j).$$

One of the main successes in the theory of rough paths was to give precise conditions on X and f for Picard iterations to converge ([3]).

1.3 Iterated integrals

Iterated integrals are integrals of the form:

$$X_{s,t}^{(\tau)} = \int \dots \int_{s<u_1<\dots<u_k<t} dX_{u_1}^{(\tau_1)} \dots dX_{u_k}^{(\tau_k)}$$

where $\tau = (\tau_1, \ldots, \tau_k)$ is called a word, with letters $\tau_i \in \{1 \ldots n\}, i = 1, \ldots, k$. By definition, integrating an iterated integral produces an iterated integral:

$$\begin{aligned} \int_0^T X_{0,s}^{(\tau)} dX_s^{(j)} &= \int_0^T \int \ldots \int_{0<u_1<\ldots<u_k<s} dX_{u_1}^{(\tau_1)} \ldots dX_{u_k}^{(\tau_k)} dX_s^{(j)} \quad (6) \\ &= X_{0,T}^{(\tau_1,\ldots,\tau_k,j)} \quad (7) \end{aligned}$$

If X is a geometric p-rough path, *i.e.* it can be approximated by paths of bounded variation (see [3] for a precise definition), then the integrals obey the usual integration-by-parts rule, which can be generalized as follows:

$$X_{s,t}^{(\tau)} X_{s,t}^{(\rho)} = \sum_{\alpha \in \tau \sqcup \rho} X_{s,t}^{(\alpha)} \quad (8)$$

The shuffle product $\sqcup$ of two words τ and ρ is the set of all words using the letters in τ and ρ such that they are in their original order. For example $13 \sqcup 42 = \{1342, 1432, 4132, 1423, 4213, 4132\}$ but 1324, for example, does not belong to the set.

This result holds only in the case of deterministic drivers X, or Stratonovitch integrals. A similar relation exists for Itô integrals but requires a small correcting term ([5]) making them slightly less practical in this context. A simple transformation allows diffusion processes to be defined with respect to Itô or Stratonovitch integrals ([1]):

$$dy_t = \mu(y_t)dt + \sigma(y_t)dB_t \Leftrightarrow dy_t = \left(\mu(y_t) - \frac{1}{2}\sigma'(y_t)\sigma(y_t)\right)dt + \sigma(y_t) \circ dB_t$$

It immediately follows from equation (8) that any power of an iterated integral is a sum of iterated integrals and that a polynomial of iterated integrals is also a linear combination of single iterated integrals. For example:

$$\begin{aligned} X_{0,T}^{(1)} X_{0,T}^{(2,3)} &= X_{0,T}^{(1,2,3)} + X_{0,T}^{(2,1,3)} + X_{0,T}^{(2,3,1)} \\ \left(X_{0,T}^{(1)}\right)^2 &= X_{0,T}^{(1)} X_{0,T}^{(1)} \\ &= X_{0,T}^{(1,1)} + X_{0,T}^{(1,1)} \\ &= 2X_{0,T}^{(1,1)} \\ \left(X_{0,T}^{(1)}\right)^\ell &= \ell!\ X_{0,T}^{(1,\ldots,1)} \\ X_{0,T}^{(0,1,0)} X_{0,T}^{(1,1)} &= X_{0,T}^{(0,1,0,1,1)} + 2X_{0,T}^{(0,1,1,0,1)} + 3X_{0,T}^{(0,1,1,1,0)} \\ &\quad + X_{0,T}^{(1,0,1,0,1)} + 2X_{0,T}^{(1,0,1,1,0)} + X_{0,T}^{(1,1,0,1,0)} \end{aligned}$$

The number of terms from the product of iterated integrals grows fast, exponentially if all letters are different.

1.4 Expansions

We are now in position to derive expansions for the solution of a differential equation. Picard iterations yield:

$$\begin{aligned}
Y_{0,T}(0,j) &= 0 \\
Y_{0,T}(1,j) &= \int_{0,T} f_{j1}(Y_s(0))dX_s^{(1)} + \ldots + \int_{0,T} f_{jn}(Y_s(0))dX_s^{(n)} \\
&= f_{j1}(Y_0)X_{0,T}^{(1)} + \ldots + f_{jn}(Y_0)X_{0,T}^{(n)} \\
Y_{0,T}(2,j) &= \int_{0,T} f_{j1}(Y_s(1))dX_s^{(1)} + \ldots + \int_{0,T} f_{jn}(Y_s(1))dX_s^{(n)} \\
&= \int_{0,T} f_{j1}(Y_{0,s}(1) + Y_0)dX_s^{(1)} + \ldots + \int_{0,T} f_{jn}(Y_s(1) + Y_0)dX_s^{(n)}
\end{aligned}$$

Since $Y_{0,s}(1,.)$ is a sum of iterated integrals, the polynomials $f_{ji}(Y_s(1)+Y_0)$ are also sums of iterated integrals and so is their integration with respect to $X^{(i)}$. $Y_{0,T}(2)$ is thus a sum of iterated integrals and by recursion all $Y_{0,T}(r,.)$ are. A formal proof in the context of differential equations driven by rough paths can be found in [4].

Example

Consider the Ornstein-Uhlenbeck process: $dy_t = a(1-y_t)dt + bdW_t$ and $y_0 = 0$. In this case, $X_t^{(1)} = t$, $X_t^{(2)} = W_t$ and $Y_0^{(i)} = 0$ for $i \in \{1,2\}$. Applying Picard iterations, we obtain:

$$\begin{aligned}
Y_{0,T}(0) &= 0 \\
Y_{0,T}(1) &= \int_0^T a(1-0)dX_s^{(1)} + \int_0^T bdX_s^{(2)} \\
&= aX_{0,T}^{(1)} + bX_{0,T}^{(2)} \\
Y_{0,T}(2) &= \int_0^T a(1-(aX_s^{(1)} + bX_s^{(2)}))dX_s^{(1)} + \int_0^T bdX_s^{(2)} \\
&= aX_{0,T}^{(1)} - a^2X_{0,T}^{(1,1)} - abX_{0,T}^{(2,1)} + bX_{0,T}^{(2)} \\
Y_{0,T}(3) &= aX_{0,T}^{(1)} - a^2X_{0,T}^{(1,1)} + a^3X_{0,T}^{(1,1,1)} + a^bX_{0,T}^{(2,1,1)} + abX_{0,T}^{(2,1)} + bX_{0,T}^{(2)}
\end{aligned}$$

The solution of the stochastic differential equation can thus be approximated by a series of iterated integral of the drivers, whose coefficients are a function of the parameters. The iterated integrals capture the statistics of the drivers and are separated from the parameters.

This derivation can be readily implemented in `Mathematica` ([6][1]) (see [5] for an implementation of the shuffle product) but suffers a major drawback: each product of iterated integrals being a shuffle product, the number of terms produced grows extremely fast (exponentially in the worst cases) and rapidly becomes unmanageable. In the next section, we introduce a re-parametrisation

[1]The code will work with `Mathematica` version 7, when parallel computing was introduced.

of the problem that circumvents this problem by providing an alternative representation of the expansion which can be processed in a distributed manner, alleviating large memory requirements.

2 Re-parametrisation of the polynomials

One thing to note in the derivation of expansions is that each successive iterations requires the explicit linear combination of iterated integrals for the previous iteration; evaluating $Y_{0,T}(r)$ requires the complete expansion for $Y_{0,T}(r-1)$. As the number of terms grows, manipulating this object rapidly becomes unwieldy.

In this section, we describe a re-parametrisation of the polynomials which bypasses the need for an expansion in terms of iterated integrals. It provides a more compact representation of the approximate solution and is naturally amenable to parallel processing. For clarity of exposition, we first introduce the approach in the case of a one-dimensional ($m = 1$) differential equation with n drivers. We assume the polynomials f_{1i} to be of degree less or equal to q. In the last subsection, the procedure is generalized to m-dimensional differential equations.

2.1 One-dimensional case

We first remark that a polynomial $P(y)$ can be written in terms of $y - y_0$ by writing its Taylor expansion around y_0:

$$P(y) = \sum_{k=0}^{q} \frac{1}{k!} \partial^k P(y_0)(y - y_0)^k \tag{9}$$

Next we introduce the following new operation for iterated integrals:

$$X^{\alpha}_{s,t} \triangleright X^{\beta}_{s,t} = \int_s^t X^{\alpha}_{s,u} dX^{\beta}_{s,u} = \int_s^t X^{\alpha}_{s,u} X^{\beta_-}_{s,u} dX^{\beta_{\text{end}}}_u \tag{10}$$

where β_- is the word β with the last letter removed and β_{end} the last letter of β. This is a non-associative, non-commutative operation – in fact, it can be viewed as a non-commutative dendriform. We can rewrite the Picard iteration using this operation. Note that from now on, the interval $[s, t]$ will be fixed to

$[0,T]$ and will be omitted.

$$\begin{aligned} Y(0) &= y_0 - y_0 = 0 && (11)\\ Y(1) &= \sum_{i=1}^{n} \int_0^T f_{j1}(Y_s(0)+y_0)dX_s^{(i)} && (12)\\ &= \sum_{i=1}^{n} f_{j1}(y_0)X^{(i)} && (13)\\ Y(r+1) &= \sum_{i=1}^{n} f_{1i}(Y_s(r)+y_0) \rhd X^{(i)} && (14)\\ &= \sum_{i=1}^{n}\sum_{k=0}^{q} \frac{1}{k!}\partial^k f_{1i}(y_0)Y_s(r)^k \rhd X^{(i)} && (15)\\ &= \sum_{k=0}^{q} \left(((Y_s(r))^k \rhd \sum_{i=1}^{n} (\partial^k f_{1i}(y_0)X^{(i)}) \right) && (16) \end{aligned}$$

Therefore, if we define the objects Q as:

$$Q^k = \sum_{i=1}^{n} \frac{1}{k!}\partial^k f_{1i}(y_0)X^{(i)}, \tag{17}$$

the Picard iteration takes the following form:

$$\begin{aligned} Y(1) &= Q^0\\ Y(r+1) &= \sum_{k=0}^{q} Y(r)^k \rhd Q^k \end{aligned}$$

where $Y(r)^k$ is the usual product.

2.2 Description of the approach

Consider a one-dimensional differential equation with quadratic functions f_{1i}, *i.e.* $q=2$. The new representation yields:

$$\begin{aligned} Y(1) &= Q^0 && (18)\\ Y(2) &= 1 \rhd Q^0 + Q^0 \rhd Q^1 + (Q^0)^2 \rhd Q^2 && (19)\\ &= Q^0 + Q^0 \rhd Q^1 + (Q^0)^2 \rhd Q^2 && (20)\\ Y(3) &= Q^0 + (Q^0 + Q^0 \rhd Q^1 + (Q^0)^2 \rhd Q^2) \rhd Q^1 && (21)\\ &\quad +(Q^0 + Q^0 \rhd Q^1 + (Q^0)^2 \rhd Q^2)^2 \rhd Q^2 && (22) \end{aligned}$$

where $(Q^k)^\ell$ is the k^{th} object Q to the power ℓ. Crucially, this new representation does not require the explicit computation of the shuffle products, keeping the number of terms under control. Moreover, the expression can be expanded into its summands and each summand be processed independently. Thus, we avoid the handling of a large expression and are able to parallelize the computation.

Note that the representation only depends on the maximum degree of the polynomials but not on the number of drivers. The Picard iteration can therefore be done only once, stored in a file and used at a later date for any system that uses the same maximum degree q.

Using this representation, it is possible to derive the stochastic expansions through a few stages:

1. Expand the expression $Y(r)$ into its mononials u. Each monomial u is a function of Q's that uses the non-commutative products. Importantly, the objects Q and the product $\triangleright$ are only used as place-holders at that stage. For example, the first three monomials for $Y(3)$ are Q^0, $Q^0 \triangleright Q^1$ and $(Q^0 \triangleright Q^1) \triangleright Q^1$.
2. For each monomial u, the objects Q are replaced by their values from the model at hand. Each Q is a weighted sum of the drivers $X^{(i)}$ (eq.17), so each u becomes a polynomial V of $X^{(i)}$ in terms of the non-commutative product. As in the previous step, the product is still only employed as a place-holder and not instantiated.
3. Each polynomial V is expanded into its monomials v. Each v is a function of $X^{(i)}$ and $\triangleright$.
4. For each monomial v, the product is instantiated; its actual definition in terms of shuffle product is only used at this later stage.

Each process only requires a fraction of the memory that a direct approach (replacing Q's by the $X^{(i)}$ and using the product's definition) would. Moreover, at all stages, each monomial can be processed independently from the rest, leading to natural parallelization. It is also important to note that the whole expression is actually stored in a file and thus not in memory at any point. The exact details of this procedure are given in section 3.

2.3 Generalization

In the multidimensional case, the Taylor expansion of a polynomial requires a larger number of terms that involve cross-products between the components of the vector Y. The objects Q are not indexed by $k \in \{0, \dots, q\}$ but by the set $\mathrm{OW}_m(0, q)$ of the ordered words of length up to q written with letters $\{1, \dots, m\}$ and are now defined as:

$$Q_j^\tau = \sum_{i=1}^{n} \frac{|\tau|!}{c(\tau)} \partial_\tau f_{ji}(Y_0) X^{(i)} \tag{23}$$

for $j \in \{1, \dots, m\}$. The constant $c(\tau)$ is the number of different words we can construct using the letters in τ. The Picard iteration becomes:

$$Y(1)^{(j)} = Q_j^\emptyset \text{ and } Y(r+1)^{(j)} = \sum_{\tau \in \mathrm{OW}_m(0,q)} Y(r)^\tau \triangleright Q_j^\tau$$

3 Implementation

This section describes how this new approach was implemented in Mathematica. In this part, iterated integrals of the drivers ($X^{(i_1,\dots,i_n)}$) are denoted $j^{(i_1,\dots,i_n)}$ to follow convention and drivers are numbered from 0 with the first driver representing time.

3.1 Shuffle product

Since each product of two iterated integrals is a shuffle product, special care must be taken of its implementation. [5] has shown a way of writing the product in Mathematica:

```
Tocino[j[{x_}], j[a_List]] :=
  Sum[j[Insert[a, x, k]], {k, 1, Length@a + 1}];
Tocino[j[a_List], j[{x_}]] := Tocino[j[{x}], j[a]];
Tocino[j[a_List], j[b_List]] :=
    Ap[Tocino[j[a], j[Drop[b, -1]]], Last@b] +
            Ap[Tocino[j[Drop[a, -1]], j[b]], Last@a] /; (Length@a > 1 && Length@b > 1);
```

This is a direct and natural translation of the following result:

$$J^{\alpha}J^{\beta} = \int J^{\alpha-}J^{\beta}dJ^{\alpha_{\text{end}}} + \int J^{\alpha}J^{\beta-}dJ^{\beta_{\text{end}}}$$

We present a new implementation of the shuffle product. The product is done iteratively instead of recursively and is based on string transformation. To calculate the shuffle product of two words α and β, we first set the string 'aa...aabb..bb', with as many a's and b's as there are letters in α and β respectively (line 2). We then replace all occurences of 'ab' by 'ba' (line 6) and keep on iterating the replacement until none are possible (*i.e* when the word is 'bb..bbaa..aa') while keeping track of the new words generated (lines 4-6). The set of all words thus created is the shuffle product of two arbitrary words of the same lengths of α and β. Finally, the letters a and b in each word are replaced by their actual values from α and β (lines 7-9).

```
1 Shuffle[a_List,b_List]:=Module[{list,u,v},
2         u={StringJoin@Join[Table["a",{Length[a]}],Table["b",{Length[b]}]]};
3         v=Table[0,{StringLength[u[[1]]]}];
4         list=Flatten@NestWhileList[
5                         DeleteDuplicates@Flatten@
6                                 StringReplaceList[#,"ab"->"ba"]&,u,#!={}&];
7         (v[[#[[1]]&/@StringPosition[#,"a"]]]=a;
8         v[[#[[1]]&/@StringPosition[#,"b"]]]=b;
9         v)&/@list];
```

Figure 3.1 shows the relative performances of the implementation. The product of two iterated integrals with random words is calculated with the recursive definition and the iterative definition. The ratio of the time spent by the former over the latter is recorded and the process is re-iterated 1,000 times for each final-word length. On average, the iterative method is a bit slower on

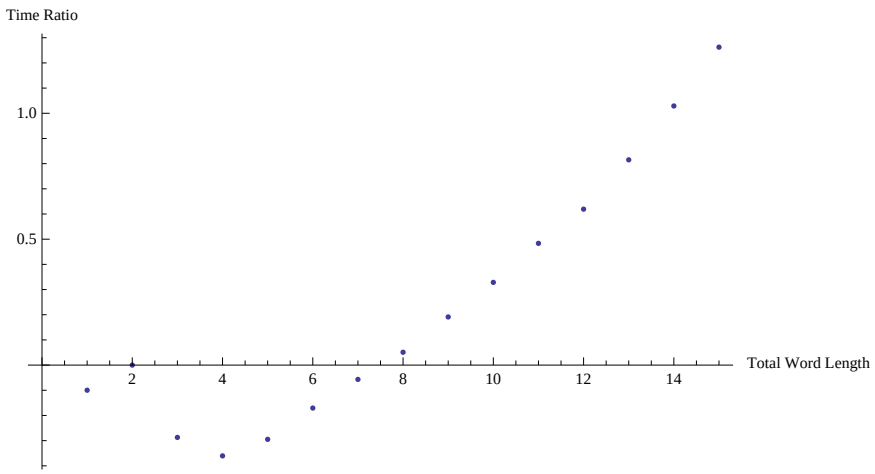

Figure 1: Log2 of the average of the relative speed of the iterative over the recursive method.

short words but faster for long words (from a total length of 8). Since longer words are more numerous, using the iterative method will be advantageous.

3.2 Non-commutative product

The non-commutative and non-associative product $\triangleright$ is implemented as $\odot$ in `Mathematica`, an operation with no built-in meaning. We only set a few basic properties for this product:

```
Unprotect[CircleDot]; (* ⊙ = esc c . esc *)
        1⊙x_ := x;
        x_⊙1 := 0;
        0⊙x_ := 0;
        x_⊙0 := 0;
Protect[CircleDot];
```

Since $\odot$ is only used as a placeholder, its actual definition in terms of shuffle product (eq. 10) is coded in another function, `NCP`:

```
        NCP[0, j[b_List]] := 0;
        NCP[1, j[b_List]] := j[b];
        NCP[j[a_List], 0] := 0;
        NCP[j[a_List], 1] := j[a];
        NCP[j[a_List], j[{}]] := j[a];
        NCP[j[a_List], j[b_List]] :=
                        Ap[j[a]*j[Drop[b, -1]], Last[b]] /; Length[b] > 0;
        NCP[n_*j[a_List], j[b_List]] := n*NCP[j[a], j[b]];
        NCP[j[a_List], n_*j[b_List]] := n*NCP[j[a], j[b]];
        NCP[n_*j[a_List], m_*j[b_List]] := n*m*NCP[j[a], j[b]];
        NCP[x_ + y_, z_] := NCP[x, z] + NCP[y, z];
        NCP[x_, y_ + z_] := NCP[x, y] + NCP[x, z];
```

```
NCP[x_*(y_ + z_), t_] := NCP[x*y, t] + NCP[x*z, t];
NCP[(y_ + z_)*x_, t_] := NCP[x*y, t] + NCP[x*z, t];
```

3.3 Picard iteration

The usual Picard iteration is implemented with a helper function `PicardIteration` as following:

```
PicardIteration[f_List,X_]:=
                Total[MapIndexed[(Ap[#1[X]*j[{}],First[#2]-1])&,f]]
Picard[f_List,X0_,n_Integer]:=
                Nest[(PicardIteration[f,#])&,X0,n]
```

For example, `Picard[f,x0,4]` outputs the stochastic approximation of the SDE with the functions f collected in a list in the first argument. This was used in [4] for a system with linear drift and quadratic variance.

With the new representation in Q, it can be written directly as in eq. 2:

```
PicardQ1Dim[Q_, R_, q_] :=
                Nest[Q[0] + Sum[(#^r)⊙Q[r], {r, 1, q}] &, Q[0], R - 1];
```

`R` is the number of iterations to be calculated and `q` the maximum degree of the polynomials f. `PicardQ1Dim[]` produces a very compact representation of the expansion, which needs to be processed further in order to give the same result as `Picard[]`.

3.4 Distributed processing of monomials

Going from the compact representation provided by `PicardQ1Dim[]` to the linear combination of iterated integrals j's is done in a few stages. Each stage modifies the representation of the stochastic expansion in such a way that a) computational requirements are minimized and b) it can be parallelized.

As described in section 2.2, the workflow goes as follows:

1. `PicardQ1Dim[]` produces a polynomial in Q and $\odot$.
2. Each monomial (in Q and $\odot$) is extracted and stored in a list (actually a file).
3. For each monomial, the Q's are replaced by their values in j (eq.17). They are now polynomials in j and $\odot$.
4. The monomials (in j and $\odot$) from each polynomial are extracted and stored in a file.
5. The product $\odot$ is instanciated in terms of the shuffle product (eq. 10). The result is a linear combination of iterated integrals j for each monomials. The stochastic expansion is the overall sum of the all those linear combinations.

As can be seen on figure 3.4, the different polynomials are successively expanded in terms of monomials, which in turn are processed independently. Since each expression is effectively broken down in small parts and dumped

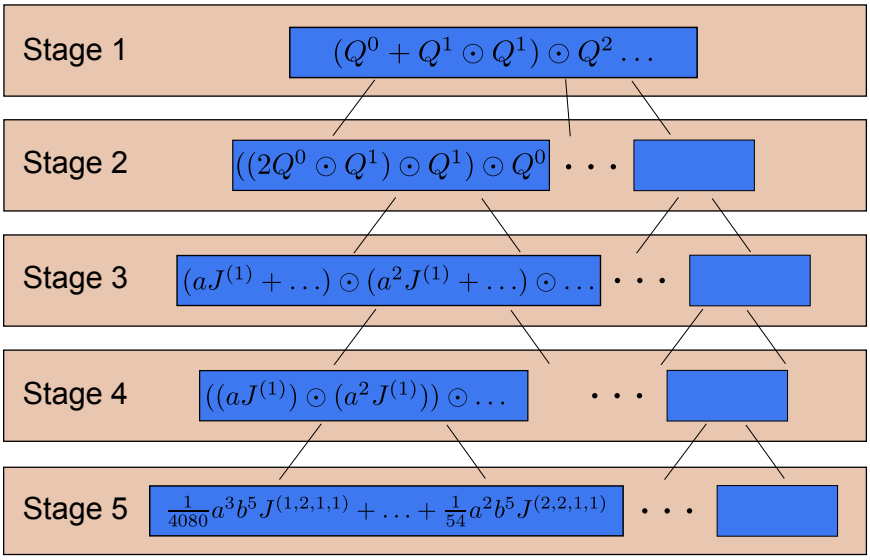

Figure 2: Workflow. From the compact representation in Q and $\odot$ to the linear combination of iterated integrals J's.

into a file, a much larger of terms can be computed, as can be seen in the next section.

Since each processing stage follows roughly the same logic, with slightly different transformation rules, we only detail the first one: going from the polynomial in Q and $\odot$ to its monomials. This is done step by step, and not by applying a global rule, since doing so would produce a very large output and run the risk of failling due to memory limits.

```
Print["=> Deriving the monomials in Q and ⊙"];
changeHappened = True;
workingFile = "uexpansion";
finalFile = "uexpansion_Final";
DistributeDefinitions[workingFile, finalFile, rulePlus, rulePower,CircleDot];
(* write expansion in *)
in = OpenWrite@mainFile; Write[in, PicardQ1Dim[Q, 4, 2]];  Close@in;
SetSharedVariable[changeHappened];
ParallelEvaluate[ Put[finalFile <> ToString[$KernelID]];];
While[changeHappened,
  changeHappened = False;
  positionList = GetStreamPositions@workingFile;
  DistributeDefinitions[positionList];
  ParallelEvaluate[in = OpenRead@workingFile];
  ParallelEvaluate[Put["temporary_" <> ToString[$KernelID]]];
  WaitAll@Table[
    With[{sp = sp},
     ParallelSubmit[
      SetStreamPosition[in, sp];
      v = Read[in, Expression];
```

```
21        If[FreeQ[v, Plus] && FreeQ[v, Power],
22         PutAppend[v, finalFile <> ToString[$KernelID]],
23         If[Head[v] === Plus,
24          changeHappened = True;
25          Scan[PutAppend[#, "test_temporary_" <> ToString[$KernelID]] &, v],
26          firstPlus = Min@Map[Length@# &, Position[v, Plus]];
27          firstPower = Min@Map[Length@# &, Position[v, Power]];
28          If[{firstPlus, firstPower} != {Infinity, Infinity},
29           If[firstPlus <= firstPower,
30            changeHappened = True;
31            Do[v = Replace[v, rulePlus, depth], {depth, firstPlus - 2, 1, -1}];
32            v = Replace[v, rulePlus];
33            Scan[PutAppend[#, "temporary_" <> ToString[$KernelID]] &, v],
34            changeHappened = True;
35            v = Replace[v, rulePower, {firstPower - 1}];
36            PutAppend[v, "temporary_" <> ToString[$KernelID]]],
37           PutAppend[v, "temporary_" <> ToString[$KernelID]]]]]], {sp,
38       positionList}];
39    ParallelEvaluate[Close@in];
40    ConcatenateFiles[tempFileNames, workingFile]];
41  ConcatenateFiles[Map[finalFile <> ToString@# &, kernelIDList], finalFile];
```

The data to process is in `workingFile`. At each step, an entry is read from this file and processed by one of the kernels (lines 20-37). If the result is a monomial, it is added to the final file (line 22). Otherwise, it still requires further processing and is added to a temporary file (lines 25,33,36,37). When all entries in `workingFile` have been processed, `workingFile` is replaced by the temporary file (line 40). The algorithm loops back until all monomials have been extracted to `finalFile`. Note that each kernel has its own temporary and final files, as writing to the same file in parallel typically results in corrupted files and missing entries. Processing an entry consists in finding the highest 'Plus' in the expression and distributing the monomials around it. In this manner, the polynomial is first decomposed into the largest polynomials, which are processed further separately.

4 Expectation of an iterated integral

It is often of interest to calculate the moments of the solution of the SDE and this can be approximated by computing the moments of the stochastic expansion. Since the expansion is a weighted sum of iterated integrals j, its expectation is simply the weighted sum of the integrals' expectations. If the drivers consist of time and Brownian motions, the expectation of an iterated integral has a simple analytic form that can be arrived at recursively ([5]).

Here we present a more direct way of calculating this quantity. Given a word α and assuming Wiener processes ([2]):

$$EJ^{\alpha}(t) = \begin{cases} 0 \text{ if } \alpha \text{ is not a sequence of 0 and pairs } mm \\ p_{\alpha}\frac{t^{q_{\alpha}}}{q_{\alpha}!} \text{ otherwise} \end{cases}$$

where $p_\alpha = \frac{1}{2}^{\frac{\#\{\alpha_i \neq 0\}}{2}}$ and $q_\alpha = \frac{1}{2}(\#\{\alpha_i \neq 0\}) + (\#\{\alpha_i = 0\})$.
Thus, for example:

$$\begin{aligned}
EJ^{(0,1,1,0,0)} &= 1/2^{2/2} t^{(3+2/2)}/(3+2/2)! &= \frac{t^4}{48} \\
EJ^{(0,1,1,0,0,1)} &= 0 \\
EJ^{(2,2,1,1,3,3)} &= 1/2^{6/2} t^{(0+6/2)}/(0+6/2)! &= \frac{t^3}{48} \\
EJ^{(2,2,0,1,1,3,3,0,0,0)} &= 1/2^{6/2} t^{(4+6/2)}/(4+6/2)! &= \frac{t^7}{8.7!}
\end{aligned}$$

This result is implemented in `Mathematica`. The expectation for a word α is then calculated in at most $|\alpha|$ steps:

```
ExpSBM[t_, j[a_List]] := Module[{i, c},
   i = Length@a;
   c = {0, 0};
   Catch[
    While[i > 0,
     If[a[[i]] == 0,
      c += {0, 1}; i--,
      If[(i > 1) && (a[[i]] == a[[i - 1]]),
       c += {1, 1}; i -= 2,
       c = {Infinity, 0}; Throw@0
       ]]]];
   (1/2)^First@c t^Last@c/(Last@c)!];
```

5 Example

Consider the following SDE: $dY_t = a(1-Y_t)dX_1 + bY_t^2 dX_2$ and $Y_0 = 0$. In this case, $m = 1$, $n = 2$, $q = 2$. The two functions f are $f_{1,1}(x) = a(1-x)$ and $f_{1,2}(x) = bx^2$.

Only two things are required from the user: the definition of the objects Q in a transformation rule, easily obtained by derivation (eq.17), and the number of Picard iterations to be computed. In this case, the three Q's are:

$$\begin{aligned}
Q^0 &= \frac{1}{0!}(a(1-0)X^{(1} + b0^2 X^{(2)}) \\
&= aX^{(1)} \\
Q^1 &= \frac{1}{1!}(-aX^{(1)} + 2b0X^{(2)}) \\
&= -aX^{(1)} \\
Q^2 &= \frac{1}{2!}(0X^{(1} + 2bX^{(2)}) \\
&= bX^{(2)}
\end{aligned}$$

Therefore, the transformation rule corresponding to this system is:

```
ruleModel={Q[0]->aj[{1}],Q[1]->-aj[{1}],Q[2]->bj[{2}]}};
```

If the first driver is time, we can follow the convention that drivers are numbered from 0:

```
ruleModel={Q[0]->aj[{0}],Q[1]->-aj[{0}],Q[2]->bj[{1}]}};
```

A small number (*e.g.* 4) of Picard iterations is sufficient for a good approximation (see [4]). An optional parameter is the maximum word length of the iterated integrals.

Running the algorithm with these parameters, we obtain the expansion, which is a weighted sum of 676 iterated integrals and already has a `ByteCount` (size) of 394'288. Its expectation is a much smaller expression:

$$aT - \frac{a^2}{2}T^2 + \frac{a^3}{6}T^3 + (\frac{1}{4}a^3b^2 - \frac{a^4}{24})T^4 - \frac{7}{20}a^4b^2T^5 + \frac{61}{360}a^5b^2T^6 + (\frac{17}{140}a^5b^4 - \frac{1}{24}a^6b^2)T^7$$
$$+(\frac{1}{192}a^7b^2 - \frac{21}{160}a^6b^4)T^8 + \frac{157a^7b^4T^9}{3024} + (\frac{43a^7b^6}{1800} - \frac{17a^8b^4}{2800})T^{10} - \frac{1}{100}a^8b^6T^{11}$$

This is confirmed by the previous implementation of Picard iteration with the simple code:

```
P0[x_] := a (1 - x);
P1[x_] := b x^2;
expansion = Picard[{P0, P1}, 0, 4];
ExpSBM[t, expansion]
```

However, if we now set the initial value to an arbitrary y_0 instead of 0, the stochastic expansion contains a much larger number of terms. It is calculated in the same manner using Q as:

```
ruleModel={Q[0]->a(1-y0)j[{0}]+by0^2j[{1}], Q[1]->-aj[{0}]+2by0j[{1}],Q[2]->bj[{1}] };
```

This results in an expansion with 10'710 unique iterated integrals and a `ByteCount` of 15'841'624. This cannot be confirmed by the simple code as it runs out of memory and crashes before completing. Note that the expansion is usually left in a file whose entries are parts of the linear combination. Thus, it is possible to, for example, compute the expectation of the expansion without having to store it in memory at any point. Moreover, while computationaly expensive, these expansions can be calculated once in a general case and saved in a file for future application without having to recompute them *de novo*.

6 Conclusion

Stochastic expansions provide a local approximation of the solution of a stochastic (or deterministic) differential equation. They can be used for a variety of applications, from simulation to parameter estimation. However, as the number of terms grows exponentially with the desired precision, they can rapidly become unwiedly to manipulate.

We presented a new way of calculating these expansions that bypasses the limitation of the usual approach, *via* a re-parametrisation of the problem and the parallelization of the computation. We have shown that in a simple example our method was able to compute the expansion when a direct approach failed. We also presented two new approaches for efficiently deriving the shuffle product

of two iterated integrals and the expectation of an iterated integral, when the drivers are time and Brownian motion.

So far, our approach has been implemented for one-dimensional differential equation[2]. However, the theoretical foundation for the multi-dimensional case is available, as presented in section 2.3. Now that the computing requirements have been alleviated, an implementation for the general case is possible. Stochastic expansions will then be available for more complex systems.

Acknowledgements

This work was funded by the EPSRC (EP/H019588/1, 'Parameter Estimation for Rough Differential Equations with Applications to Multiscale Modelling'). We would like to thank the `Mathematica` newsgroup (`comp.soft-sys.math.mathematica group`) for their help and advice on file parallelization. The code was tested on Buster, the cluster of the department of Statistics of the university of Warwick.

References

[1] P. E. Kloeden and E. Platen. Relations between multiple ito and stratonovich integrals. *Stochastic Analysis and Applications*, 9(3):311–321, 1991.

[2] Christophe Ladroue. Expectation of stratonovich iterated integrals of wiener processes. Aug 2010.

[3] Terry Lyons and Zhongmin Qian. *System Control and Rough Paths (Oxford Mathematical Monographs)*. Oxford University Press, USA, February 2003.

[4] Anastasia Papavasiliou and Christophe Ladroue. Parameter estimation for rough differential equations. May 2010.

[5] A. Tocino. Multiple stochastic integrals with mathematica. *Mathematics and Computers in Simulation*, 79(5):1658–1667, January 2009.

[6] Stephen Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, 5th edition, August 2003.

[2] `Mathematica` package available on the author's webpage: `http://go.warwick.ac.uk/roughpaths`